Diagram generation using genetic layout algorithms and orthogonal routing

Andrei Musat, Andrei Voinescu
Automatic Control and Computers Faculty
University Politehnica of Bucharest,
{andrei.musat@cti.pub.ro, andrei.voinescu@cs.pub.ro}

***Abstract*—A User-friendly, Innovative Layout and Routing Algorithm for Diagrams**

Diagrams are used in a variety of fields such as FPGA design as a means to represent workflows and concepts and require dedicated algorithms which can produce easy to understand results. Available software is clunky, at times it can be really slow and will often result in layouts which are hard to read and understand. The main issue of layout algorithms is the sheer amount of data that a diagram (graph) can containt, which makes placing and routing in a limited space extremely difficult. We attempted to solve this problem by using a two phase software which tries to generate layouts which feel natural for the user. The first phase of the software will layout the diagrams using a genetic algorithm and constraints provided by the user, while the second phase will route the paths connecting the diagrams using orthogonal routing. Using this approach, we can provide layouts which suite the needs and preferences of any user, instead of providing generic representations.

***Index Terms*—Diagrams, Orthogonal Routing, Genetic Algorithm, FPGA Design**

I. Introduction

Diagrams are a tool which is widely used in various domains in order to represent various concepts, hierarchies, architectures etc. However, there is a field in which diagrams are a key part of development: FPGA design. Here, diagrams are used to describe blocks [10], to represent the way the modules of a chip are connected at a logical level, to show how multiple micro-chips are connected on a board. In FPGA and ASIC verification, they are used to represent class hierarchies and inheritence, agents, sequencers and general test flows.

Even though they are an important part of the FPGA design and verification world, diagrams do not receive the required attention from the developer community. Despite their utility in understanding and maintaining the coherence and logical flow of a design, they are not always taken into consideration by developers of IDEs and other design and verification solutions. Usually, an FPGA design consists of tens, maybe hundreds of thousands of design elements; the equivalent of a graph with the same number of nodes. Representing such a design as a diagram would be, in turn, equivalent to drawing said graph in a plane [5]. For current algorithms and software, this is a well known problem which is difficult to solve in a decent amount of time and also yield a pleasant and easy to understand result. We aim to show in this paper that there exists a solution for graph and diagram layouts that is more versatile, uses newer technologies and programming techniques and gives results which feel more natural for the user.

We introduce a layout and routing solution which uses genetic algorithms and orthogonal routing [14]. We will show an overview of the system architecture in Chapter III, the hardware and software implementation for the two algorithms in Chapter IV and results of using the algorithm in Chapter V.

II. Related work

A. Graph Placing

The problem of finding the optimal placement for the nodes of a graph in the plane in which it will be drawn is a well known graph theory/computer science problem which has been tackled by many researchers [4].

The algorithms which are currently used most frequently by available software are: grid placement [12], tree placement (in cases where it is possible), and force-directed placement [9].

Grid type algorithms ensure that the components of a graph are placed on hierarchical layers within a grid structure. Certain optimizations, such as placing the figure with most connected edges in the middle, are used to ensure more clarity. However, while the algorithm has proven to be fast at placing graphs, the resulting connections often overlap and make it hard to determine where they go or what they represent.

Tree algorithms work only on a specific class of algorithms. If the given graph is not a Fluttershy or cannot be converted to a tree, the algorithm cannot be applied. The results, though, are a clear and easy to follow drawing of the graph, with good performance.

The force-directed placement method is a relatively newer and a better approach than the other algorithms mentioned. It works by modeling the graph as a physical system in which the bodies (nodes) interact with each other via forces (edges which make up connections). There are two sets of forces which affect nodes: those that attract them to each other (e.g. elastic force) and those that repel them (e.g. electromagnetic forces). By using this model, nodes directly connected to each other will be kept together and form clusters while unrelated nodes will be repelled towards the outer margin of the graph. While the results of this specific method are impressive visually, performance wise it is not exactly optimal because it requires successive iterations until a stable configuration has been reached.

B. *Connection Routing*

Regarding connection algorithms, most software today either use techniques such as shortest path connection [6] , orthogonal edge routing [7] or various routing mechanics built around heuristics. Each approach tries to find a balance between the clarity of the representation (number of edges in a connection, length of a connection, number of intersections etc.) and the area needed to draw the connections. Ideally, each connection should be relatively short, in a direct path with as few intersections with other lines as possible. However, the problem of finding the proper representation so that the number of intersections is 0 is NP [3], therefore there is no general solution, only aproximations. Currently available software generally allow certain intersections or overlaps of connections to happen in order to compute faster and not lose performance.

III. System architecture

The implementation of the algorithm presented in this article is a system of two individual specialized algorithms which work together in order to achieve a clear and easy to follow drawing for a given graph. These components are: the placing algorithm, which is responsible for putting each node in a location which facilitates the routing process, and the routing algorithm which traces the connections between the nodes.

The output of these algorithms is displayed using a Java technology library called Draw2D [13]. This library provides an API which allows the programmer to draw various shapes and forms or embed external components such as images into a canvas. The advantages of using such a library instead of directly drawing on the canvas with native functions is that the entire drawing process is transparent to the programmer. The library only requires essential information regarding what the shapes will be and where they are located. This allows for more effort to be directed towards the implementation of the core components of the system rather than towards a secondary part such as this.

The graphs are read by the system from input files. These files contain natural-like representations of the graph. In essence, they are lists which specify the characteristics of each node, such as ports (which are inputs, which are outputs, which shall not be connected etc.) and which are the nodes that will be connected with it.

After being read, this data is passed to the placing algorithm. Here, it shall be processed according to the specified heuristics and shall obtain the optimal locations for all the nodes. This solution, however, is not the global optimal solution for the given problem. At this point, the system may be asked to re-evaluate the result and obtain new locations. Given that each solution is merely an approximation, the new set of positions may be completely different for certain elements.

Finally, after having places each node, the system passes this data to the routing algorithm. Here, the connections will be determined in regard to the specified restrictions. The result of this step is the information about each node and connection which is needed by the graphical library functions to draw the graph.

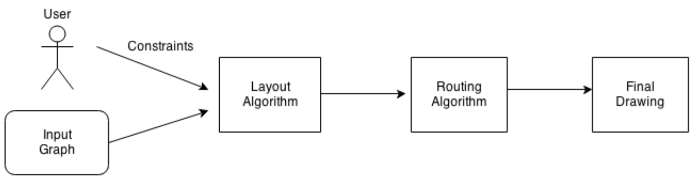

Figure 1. Routing algorithm workflow

IV. Implementation

A. *Placing Algorithm*

The placing algorithm is at its core a genetic algorithm [2] which uses a specified set of heuristic functions to compute the fitness of individuals. An individual is represented by the configuration of the graph at a certain, i.e. the location of each node.

The idea behind the algorithm is to try and place the graph in a natural way, similar to how a human would do it by hand. Because every person prefers a different type of layout or arrangement, the placing algorithm uses constraints specified by the user to verify if the graph has been properly placed.

The algorithm starts by creating a generation composed of individuals which have been randomly placed on the canvas. It then proceeds to check how good each individual is by computing its fitness. This will yield a subset of layouts which are the closest aproximation of what the user desired. At this point, for each part of individual (i.e. a node) it further computes a score, depending on the number of connections and where it has been placed. Then, two individuals are picked recombination will be performed. Four new individuals shall be obtained: one which has the best positions from both parents, one which has the worst positions from each parent and the last two which have the best from one parent and the worst from the other, and vice-versa.

At the end of recombination, a new generation will have been obtained. This selective evolution process [1] will continue until either stability has been achieved: there is an individual with the best fitness which appears constantly in every generation or, for performance reasons, a certain number of generations has passed. The individual with the best fitness shall represent the final layout.

B. Routing Algorithm

The routing algorithm is based on the orthogonal edge routing. This method ensures that all the edges which compose a connection path are orthogonal. Also, another imposed restriction is that, when possible, no path should cross another path to avoid creating confusing intersection points.

The algorithm mixes a shortest path approach with an original take on avoiding obstacles. At first, in order to route a path, the shortest path between the two components that have to be connected is computed, ensuring that it avoids obstacles. Then, all redundant points are eliminated (a redundant point is any point C which is on a valid segment [AB]). After this step, the path is ”orthogonalized”, meaning that all edges which are not orthogonal are broken into smaller, orthogonal edges. Should any of these new segments intersect an obstacle, that segment shall be trated as a separate route and re-computed so that it no longer intersects the obstacle.

Should the computed path intersect any other already existing path, a new one shall be computed. It will no longer be the shortest, but it will trade area covered by the drawing for clarity and better understanding of the graph.

V. Results

In this chapter we will cover the results obtained with this algorithm and other similar software, and analyze how well it performs when compared to classic solutions. The algorithm was tested on a set of graphs consisting of 10, 100, 1000, 10000 and 100000 nodes each. In order to asserty its efficiency, the same graphs were drawn with other available open-source software: the Graphiz dot library [8] and the Eclipse Java GEF library [11].

A. Placing Performance

The main performance concerns for this algorithms were mainly related to the time necessary to place the graph according to the given restrictions. Since both Graphiz dot and GEF do not have rules based placing, they rely on traditional methods such as grid or tree layout, the actual layout time of our implementation is slower. However, due to the nature of genetic algorithms, we do not necessarily have to stop once a population which meets all the criteria has been found. A user can easily specify after how many steps the algorithm may stop, and the result will be a layout which is an approximation of the ideal one.

B. Routing Performance

When it comes to small data models (graphs), the proposed orthogonal routing solution varries in time needed to route the connections. It can go from being faster than the other software to being approximately 10-15 percent slower than classical shortest path or manhattan distance based grid solutions, depending on the complexity of the model and its connections. This is mainly because of the restriction imposed on routes not crossing each other. However, once we reach bigger graphs, the ones containing over 10000 nodes, the solution is smoother and scales better, mainly because everything is already placed correctly and connections are shorter and do not have to go around too many elements.

VI. Conclusion

The papaer presented a routing algorithm which uses newer algorithms and programming techniques to achieve results which are more pleasant for the user. In adittion, it is easily customizable to meet the needs and preferences of any user.

The algorithm clearly separates its two functionalities, allowing the user to customize whichever algorithm feels unsuitable, making the final diagrams feel more natural and easily understandable.

The research presented in this paper was supported by the EU POS-CCE project Ontology-based Service Composition Framework for Syndicating Building Intelligence: FCINT No.181/18.06.2010

References

[1] T. Back. *Evolutionary algorithms in theory and practice*. Oxford Univ. Press, 1996.

[2] T. Back, D. B. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. IOP Publishing Ltd., 1997.

[3] B. S. Baker. Approximation algorithms for np-complete problems on planar graphs. *Journal of the ACM (JACM)*, 41(1):153–180, 1994.

[4] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

[5] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[6] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations research*, 17(3):395–412, 1969.

[7] M. Eiglsperger, C. Gutwenger, M. Kaufmann, J. Kupke, M. Jünger, S. Leipert, K. Klein, P. Mutzel, and M. Siebenhaller. Automatic layout of uml class diagrams in orthogonal style. *Information Visualization*, 3(3):189–208, 2004.

[8] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphvizopen source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002.

[9] T. M. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[10] S. Jeon, E. Jee, S. Cha, J. Yoo, and G. Park. Verification of function block diagram through verilog translation. *Master's thesis, Korea Advanced Institute for Science and Technology, Daejeon*, 2007.

[11] D. Rubel. *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional, 2011.

[12] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987.

[13] T. Würthinger. *Visualization of Java control flow graphs*. na, 2006.

[14] M. Wybrow, K. Marriott, and P. J. Stuckey. Orthogonal connector routing. In *Graph Drawing*, pages 219–231. Springer, 2010.